Using simulation in an objects-early approach to CS1 and CS2

Owen Astrachan
Duke University
ola@cs.duke.edu

Claire Bono
University of Southern California
bono@usc.edu

(listed alphabetically)

Abstract

Our philosophy is that CS1/CS2 students learn best as *apprentices*, by studying the work of more experienced program designers and writers and by extending and modifying programs provided by these experienced programmers. Object-oriented programming supports this philosophy well and permits our students to explore a realm of programs not possible using a more traditional "structured" approach. Our approach emphasizes using objects before creating them, reading and modifying programs before designing them. Although our philosophy transcends a particular object-oriented language, we have chosen C++ for our courses; our approach facilitates its use by novices. In this paper we discuss our approach, and suggest simulation as a rich area for case studies, labs, and assignments when employing this *objects early* approach. We provide details of two simulations we have used in our courses: a simulation of the card game *war* and a queueing simulation.

1 Introduction

Object-oriented is a new buzzword and a "hot" area for software engineers, computer scientists, and by extension for students of computer science. We believe that the object-oriented approach has much to offer and should strongly be considered for use in the introductory sequence of computer science courses (traditionally CS1, CS2, and CS3). We approach object-oriented programming as evolutionary rather than revolutionary — based on organizing programs around modules and abstract data types that has been used in such courses for many years. Rather than use languages such as standard Pascal that have little support for a modular approach, we have chosen to use C++, a language that provides support for the structured programming methodology as well as support for an object-oriented approach. Our approach to using objects extends an approach to using procedures espoused in [13] in which Pattis warns that:

> *It is disingenuous to extol the advantages of subprograms to our students, when they find it harder to write simple programs using subprograms — because they are not familiar with abstraction, but more importantly because using subprograms truly makes simple programs harder to write.*

Just as Pattis warns that it can be a mistake to adopt a wholesale *procedures early* approach, we do NOT advocate exclusive use of *objects early*. In object-oriented terminology, students must become clients of objects, modules, and libraries supplied to them before they can be expected to create their own. There are certainly other ways to use an object-oriented approach early in the curriculum; in particular the emphasis may turn to designing objects as opposed to designing programs[1]. However, just as students often fail to see the usefulness of procedures even when coerced to do so (we have probably all seen assignments that mandate "at least three procedures must be used"), we believe that the usefulness of objects and an object-oriented approach is best seen by having students modify and extend code, objects, and libraries with which they are supplied before writing and designing on their own. In addition, by

[1] The authors were participants in a recent NSF workshop for those using object-oriented techniques early in the curriculum. The group was divided as to whether the design or use of objects should be emphasized from the start of a CS1 course.

supplying our students with appropriate classes and code we are able to introduce more complex programs earlier in the curriculum.

In addition to these philosophical reasons for employing an object-oriented approach, we have been led by pragmatic reasons as well. Many schools are beginning to use C in the introductory sequence of courses [15] because it is used later in the curriculum and because industry demands it. At each of our institutions C is or has been used in the first course for computer science majors. Students in this course have a great deal of difficulty in mastering the idiosynchracies of C for reasons discussed in [11].

Thus pragmatics and philosophy have led us to adopt C++ as the language in all introductory courses at Duke and in the CS3 course at USC. Although we view C++ as a much better C, it is much more than that. C++ also provides good support for teaching abstraction, encapsulation, and inheritance. It allows us to supply our students with many software components — classes and functions — that they use in modifying, enhancing, and developing programs. It may not be the ideal language for use in the introductory sequence, but its increasing popularity and the demand by industry for students versed in object-oriented programming make it a very attractive language for our use.

In the following sections we discuss further how to use an object-oriented approach early supported by C++ and how simulation fits well with that approach. We then provide concrete examples of two different simulations that reflect our methodology of reading, using, and modifying objects before creating them. These examples will give a flavor for the case studies and assignments that we provide our students.

2 Philosophy

It is our experience that most students do not become proficient at designing programs in CS1 (and not often in CS2). This is not surprising considering the difficulty of program design. Consider the comparison of a computer program and an English essay. Our colleagues in first-year writing courses often despair at the ability of their students to craft an essay — this after at least five years of practice before arriving at college. Is programming so simple that we expect students to be proficient after one semester?

Our philosophy is that CS1/CS2 students learn best as *apprentices*, by studying the work of more experienced program designers and writers and by extending and modifying programs provided by these experienced programmers. Students use instructor-supplied modules, i.e., classes and subroutines, that enable them to work with much larger programs, that do more interesting things, than they would be able to if all programming assignments were based entirely on student-written code. Objects and code libraries supplied to the students are often part of a complete program, i.e., a case study, that the students read and modify. This use of an "expert" solution better reflects real-world programming at the level that students may encounter if they pursue careers that involve programming. It also emphasizes the practices of code re-use and enhancement, which is facilitated by classes and inheritance — key features of object-oriented programming. Case studies [9], combined with object-oriented programming, are very good at introducing programs using the "read-call before write" model that we emphasize and that is elucidated in [12, 13].

These instructor-supplied classes and functions can be application-oriented, as described by Pattis [12], or of a more general purpose nature. One advantage of general purpose classes is that they permit students to work with some abstract data types (e.g,. stacks, queues, and trees) early in the introductory sequence since use is divorced from messy pointer manipulation and dynamic memory allocation. This use of higher levels of abstraction is one of the appeals of the approach used in Scheme-based courses [1]. We use classes in C++ to separate the programmer from the machine[2]. Later in CS1 and in CS2 students can explore the implementation of these classes, for example, by changing the representation of an abstract data type to make it more efficient.

The use of object-oriented techniques in C++ allows us to address the need for C in courses later in our curricula. Roberts [15] discusses using instructor-provided packages for the purpose of hiding the complexity of C in a first course. C++ is a much better solution since it includes call-by-reference and an I/O stream package that is extensible to user-defined objects. As another example, consider strings in standard C. In order to use string variables or parameters, students must know about pointers, about

[2] Although for better or worse, this will not turn C++ into a dynamically typed, interpreted language with garbage collection.

arrays, and be subject to discomforting error messages such as *segmentation violation* (or worse, get incorrect results) when accessing an out-of-bounds array cell. With a well-designed *String* class, students can use strings early in a course since the class will manage memory and provide a more bullet-proof semantics for parameter passing via a copy-constructor. For students, in this case ignorance is bliss. Even when a working knowledge of C is one of the goals of a first or second course, we believe it is an easy step down from C++ to C.

The approach presented here gives students a gradual introduction to design via experience. Eventually, in addition to the programming activities mentioned previously, students can be involved in writing programs "from scratch." If code or designs from previous programs are reused, all the better. In our experience students are often initially reluctant to use code supplied by others. By the end of our courses, however, the success of this approach is evident both in student evaluation and in student performance.

3 Simulation

Simulation is well-suited to an object-oriented approach, provides an interesting range of programming applications that require a computer[3], and provided the historical impetus for the design of one of the first object-oriented programming languages *Simula* [4].

Simulation has always been an important topic in computer science: the earliest computers were used for deterministic simulations of ballistic trajectories. The need for sophisticated simulations has increased at least as fast as has the power of desktop machines and supercomputers. The scale of simulations such as BBN's distributed military simulation SimNet demands software techniques that may be provided by the object-oriented paradigm. As object-oriented languages and techniques are integrated into introductory computer science courses, they facilitate to a large degree the study and use of simulation in these courses; similarly, simulation provides a rich and interesting application area for discussing and practicing object-oriented techniques.

In addition to being an important experimental technique in many other fields, simulation is also used in computer science itself as an experimental technique for evaluating the performance of computer systems, for verifying hardware designs, and as a testbed for software that will be run on "virtual" hardware — that is hardware in the design and development phase. Thus, simulation can be a way to include experimentation in CS1 and CS2 labs, as well as being useful as part of a "breadth-first" introduction to computer science. Using simulation in a classroom setting later in the curriculum is discussed in [14, 16]. Overviews of different kinds of simulation can be found in Whitney [17] and [10].

In this paper we provide detailed descriptions of two simulations we have used with great success in our classes: a simulation of the card game *war*, and a discrete event queueing simulation. We elaborate on our approach to using each of the simulations in the next two sections.

4 Simulating the Card Game War

War is a simple card game learned by almost all children in the United States. In three years of using the game as the basis for discussion in more than 10 different classes, over 95% of the students polled knew the game[4]. Because the game requires no strategy, it is useful in learning about numbers, following rules, and playing cards. Perhaps its simplicity endears it to young card-players — the simplicity also makes it relatively easy to write a program to simulate the game.

War is a two-player game, the rules follow:

1. Play begins with a deck of 52 cards that is shuffled and evenly divided between the two players. Each player keeps this pile "face down", i.e., the rank and suit of the cards are not visible. Play, described below, continues until one player runs out of cards.

[3] Generally we try to avoid programming assignments that are more suited to pencil-and-paper or hand-held calculator solutions.

[4] Some students often claim to be "experts" in war, winning games with only one pass through the deck.

2. A play consists of each player simultaneously flipping the card that is on top of the player's pile revealing the rank and suit. The player whose card has the higher rank wins both cards. If the cards are the same rank then a "war" ensues:

   (a) Each player puts three cards face down and a fourth face up in sequence according to the cadence of the words "*I de-clare war!*" (the face up card is flipped on the enunciation of the word "war"). The rank of the face up card determines the war winner, this player takes all ten cards – the original two equal cards plus the eight war cards.

   (b) If the face-up cards are of equal rank another war ensues, this process is repeated until a winner emerges.

3. When a player's pile is used up, the cards won during the playing process are shuffled and become a "new" playing pile.

4. The game is over when one player has all the cards, this player is declared the winner.

These directions are usually enough to get people started playing the game, but the specifications are not complete. What happens, for example, if a player does not have four cards when a war takes place? Most people arrive at a solution of declaring a war with either four cards or as many as possible if four cards are not available. What happens if a player's last card is equal in rank to the opponent's card? As veterans of many wars, we do not recall this happening to us; but it is certain to happen in a computer simulation — especially if the case is not covered by the code.

The simulation is based on classes for a card, a deck of cards, a hand of cards, and a player. In later courses these classes can be re-implemented, extended, and modified. The classes that support the game serve as a small, but complete example illustrating many of the techniques (and idiosyncrasies) inherent in the use of C++ to support object-oriented programming and design. The book by Budd [3] includes code for implementing a game of solitaire, but the classes used here are substantively different.

4.1 Use of the Program

Lack of space precludes inclusion of all the questions we ask of students when studying this simulation and the iterative refinement of the full program from the specification of the classes and small example programs.

Several interesting questions arise, however, in experimenting with and modifying the programs that are used to simulate war. Some of the most interesting questions are:

1. What is the average (expected) number of card-flips in a game of war? How is this number affected by the number of cards dealt? (e.g., if each player is dealt half of a deck or a complete deck.) [*We expect students to be able to determine that the number of card-flips is* $O(n^2)$ *where* n *is the number of cards initially dealt.*]

2. How does shuffling the discard pile affect the outcome of the game? For example, is it possible for a never-ending game to result? [*If the discard pile is not shuffled then it is both possible and likely that a never-ending game results. This can serve as an introduction to the halting problem.*]

3. In a trial of 100,000 "war runs", will any one war generate more than two successive wars? What is the maximum number of such wars generated?

4.2 Development of the Program

We typically use this simulation as an introduction to arrays. Students modify a program we provide, they do not need to understand how the classes for a deck and hand of cards are implemented (a hand is a subclass of a deck that supports addition of cards, decks only permit cards to be "dealt"). After modifying the simulation, we investigate the implementation of these classes (and a *String* class) as a means of introducing arrays as collections of homogeneous objects supporting random access. The design of the classes fosters discussion of object-oriented techniques.

We provide students with a skeleton program that deals cards to two players, but does not implement a real "war". The aspect of the simulation most prone to error is simulating a war (when two cards are equal). There are many special/boundary situations that must be managed as discussed above. In the initial program a war-winner is decided by a virtual coin flip. This permits early experimentation with the simulation. The development of the war routine proceeds in steps with normal (four card) wars implemented first and with exceptions dealt with by arbitrarily declaring a winner. This process continues until a full-featured version of the program is developed.

4.3 Class Discussion

We talk briefly about the implementation of the card class when the program is introduced. It is used to reinforce simple concepts such as private/public member functions and to introduce enumerated types (used to represent rank and suit). The classes for a deck and a hand are studied to understand their interface and as a simple example of inheritance as discussed above.

Later in the course, or in a CS2 course, the similarity of hands and decks to a stack can be investigated. A hand and a deck are very similar to a stack, except that the shuffle operation requires random access. In a CS2 class we might ask our students to implement a shuffleable deck using only stacks (with two extra stacks and a random-number generator a stack can be shuffled).

4.3.1 Extending the Simulation

To emphasize the concept of code re-use, we can ask students to implement a simulation of another card game called *pump the well dry*. This game involves alternate rather than simultaneous "card-flips" with the high card winning as in war, and can be implemented using the deck, hand, and player classes from war. However, honor cards (jack, queen, king, and ace) result in different play with the rank of the honor card determining how many chances the opposing player has to play another honor card which can then win the hand. This process can continue just as one war can result in another. As in war, the game continues until one player has all the cards.

The classes can also be used to implement a game of solitaire, especially if a graphical card display is provided. Different solitaire games are investigated in [3] and in [8].

5 Queueing Simulation

While war is a good example of a simulation case study for use early on in CS1, an event-driven queueing simulation is more appropriate for use in CS2. It provides a realistic application of FIFO queues and priority queues, two important abstract data types whose interface and implementations are studied frequently in data structures courses. However, it could be used in a CS1 course where students are clients of the queue classes and study the interfaces of these classes rather than their implementations. Discrete event simulations are also are a good example of event-driven programming: essential for understanding the behavior of simulations, but used more generally in many object-oriented systems, such as client-server based concurrent systems and graphical user interface frameworks.

A canonical example of a discrete event simulation is a bank with one or more tellers: customers arrive periodically, get in line, eventually get served, and then leave. The simulation allows one to derive statistics such as the average time a customer waits in line and the average line-length. By giving a simulation different parameters, e.g., changing the number of tellers, or making more extensive changes to the simulation, e.g., one line per teller vs. one common bank line; one can draw conclusions about and try to optimize the real-world scenario being simulated.

In our use of this simulation, students are provided with a multi-teller multi-line bank simulation program. Its design is loosely based on the one used in on the instructor's manual for the Abelson and Sussman text [2] and includes classes for an event, an event queue, a customer, a customer generator, a server, and a customer line, as well as the more general purpose classes FIFO queue and priority queue. The event queue is a priority queue where the insert operation corresponds to scheduling a simulated event to occur, and the delete-minimum-element operation corresponds to removing the next chronological event

for processing. The customer generator class generates customer arrival events. The customer line class contains a FIFO queue and a server as well as the necessary information for gathering statistics about the line. Inheritance is used to distinguish between different types of events: there is an ArrivalEvent class and a DepartureEvent class which both respond to "process" messages sent to them by the event-list.

There are several types of assignments that can be fashioned around this bank simulation. We classify these into data structures assignments, program modifications, and creating new simulations. These are describe in some detail below.

5.1 Data structures

As we have mentioned, event-driven queueing simulations usually use FIFO queues and priority queues. It has been our experience that at the point in our courses at which we introduce this simulation, students are not usually mature enough to design their own classes for such data structures. They can, however, be given an interface specification whose implementation they develop. We give our students such a specification and ask them to develop the implementation and to create a driver (test) program for their code. Later, if this code is used in a simulation program, the students will see the benefits of using incremental development. Students can also be asked to replace one implementation for another in the bank simulation, e.g. replace a priority queue represented linearly with a heap representation, and then compare the performance of the two versions empirically.

5.2 Program Modifications

In addition to program modifications related to traditional data structures discussed above, we give programming assignments designed to show how a well designed object-oriented program can facilitate enhancements. As a secondary goal, such tasks also motivate the students to study and try to understand the code and overall program design. Some examples of the types of assignments that can be given are:

1. *Replace the multi-server multi-line version with one that just has one line feeding all the servers. Compare the results for customer waiting time and server idle time to the original version.* This modification can be made with minimal changes to the bank simulation. The primary modifications necessary are changes to the global data structures, i.e., there is only one line instead of an array of lines, but all the code for computing statistics about lines and servers remains unchanged. There are a few other changes necessary, such as the addition of code for choosing an idle server when a customer walks up to an empty line, and dealing with the customers currently being served (they can no longer be considered part of the line).

2. *Change the simulation to model a grocery store where customers shop for some number of items before getting in line.* This involves adding a new EnterStore event class (derived from the general Event class), which will be distinct from the current Arrival event (which will mean arriving at the checkout lines in the new simulation).

3. *Introduce customers that have different behavior, and compare the results of these customers in the same simulation. For instance, in the current version of the program customers choose a line at random. Add customers that choose the shortest line, and customers that reevaluate the situation at one minute intervals, possibly switching lines several times.* This can be modeled well by making customer an abstract base class, and creating sub-classes for each of the customer types who have their own method for doing the get-in-line operation. The customer generator can be modified to generate both kinds of customers, or different customer generators can be made for each kind (with a class hierarchy that parallels the customer class hierarchy).

4. *Add a graphical display of the main entities being simulated to help understand what's happening in the simulation.* The graphical display can be incorporated into the program by making "graphical" subclasses of the server and the customer line classes which re-display themselves whenever they change state. It is a good exercise in software engineering to require students to maintain two

variants of the program, a graphical and non-graphical version, trying to keep as much code as possible common to the two.

5.3 Creating a new simulation

After studying the bank simulation in class and/or through doing some of the exercises mentioned above, students can build a different simulation application as a larger project. It is desirable that students reuse as much of the design and actual code as possible from the bank simulation in creating their own simulation. Goldberg and Robson [5] present several examples of simulation problems. They present a ferry simulation involving cars queued up at ferry terminals and ferries going between the terminals, and a car wash simulation which includes customers asking for one or more of wash, wax, and dry, and waiting in queues for each. Several textbooks also present simulation problems: Kruse [7] presents a simulation of planes landing and taking off at an airport runway and Horstmann [6] presents a simulation of a factory that includes producers and consumers.

6 Conclusions

Just as the information hiding in object-oriented programming is used to hide complexity in large systems, we are using it to reduce some of the complexity of teaching the introductory courses: to allow students to work with more interesting programs by hiding some complicated parts in classes, to make C++ nicer and simpler for the novice, and to introduce data abstractions early whose implementations will be discussed in later courses. In addition, we believe that by having students work with objects from the start, they won't develop bad habits when designing larger programs in later courses.

We have had success using simulation to teach object-oriented programming. The simulations discussed in this paper have been used for several years in various lower-division classes for both non-majors and majors taught by the authors[5]. In addition to those presented here we have successfully used other kinds of simulations, including random walk simulations, and cellular automata (e.g., Conway's Life). Our experience has been that interested students will work harder and better and that simulations engender interest. They provide us, as well as our students, with a bridge to an object-oriented approach.

References

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretion of Computer Programs.* MIT Press, McGraw Hill Book Company, 1985.

[2] Julie Sussman. *Instructor's Manual to Accompany Structure and Interpretation of Computer Programs* MIT Press, McGraw Hill Book Company, 1985.

[3] Timothy Budd. *An Introduction to Object-Oriented Programming.* Addison Wesley, 1991.

[4] O.-J. Dahl and K. Nygaard. *SIMULA 67 Common Base Proposal.* Norwegian Computing Center, 1967.

[5] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation* Addison Wesley, 1983.

[6] Cay S. Horstmann. *Mastering C++.* John Wiley & Sons, Inc., 1991.

[7] Robert L. Kruse and Bruce P. Leung and Clovis L. Tondo. *Data Structures and Program Design in C.* Prentice-Hall, 1991.

[8] Michael J. Clancy and Marcia C. Linn. *Designing Pascal Solutions: Case Studies with Data Structures.* W.H. Freeman and Co., (to appear).

[5] The complete code and assignments for both the war game and the bank simulation are available via anonymous ftp or directly from the authors.

[9] Marcia C. Linn and Michael J. Clancy. The case for case studies of programming problems. *Communications of the ACM*, 35(3):121–132, March 1992.

[10] Roger McHaney. *Computer Simulation: A Practical Perspective*. Academic Press, 1991.

[11] R.P. Mody. C in education and software engineering. *SIGCSE Bulletin*, 23(3):45–56, September 1991.

[12] Richard E. Pattis. A Philosophy and Example of CS-1 Programming Projects. In *The Papers of the Twenty-first SIGCSE Technical Symposium on Computer Science Education*, pages 34–39. ACM Press, February 1990. SIGCSE Bulletin V. 23 N. 1.

[13] Richard E. Pattis. The "procedures early" approach in CS1: A heresy. In *The Papers of the Twenty-fourth SIGCSE Technical Symposium on Computer Science Education*, pages 122–126. ACM Press, March 1993. SIGCSE Bulletin V. 25 N. 1.

[14] Richard J. Reid. Object-oriented programming in C++. *SIGCSE Bulletin*, 23(2):9–14, June 1991.

[15] Eric S. Roberts. Using C in CS1 evaluating the Stanford experience. In *The Papers of the Twenty-Fourth Technical Symposium on Computer Science Education*, pages 117–121. ACM Press, March 1993. SIGCSE Bulletin V. 25, N. 1.

[16] David J. Thuente. Simulation in the undergraduate computer science curriculum. In *The Papers of the Twenty-First Technical Symposium on Computer Science Education*, pages 53–57. ACM Press, February 1990. SIGCSE Bulletin V. 22, N. 1.

[17] Charles A. Whitney. *Random Processes in Physical Systems*. Wiley Interscience. John Wiley & Sons, Inc., 1990.